Return Values and Error Handling

Bugs, Errors, Results

bug - a flaw in a program's logic--the only solution is stopping execution and fixing the bug

- out of bounds array access, derefrencing a `nullptr`, dividing by zero, object in invalid state illegal casts
- unmet pre/post-conditions (ex: a factorial function returning a negative number!)
  - post-condition: requirement function proces to acheive upon completion

unrecoverable errors - non-bug errors where recovery is impossible, and the program must shut down

- out of memory error, network host not found, full disk

recoverable errors - the program may continue the execution

- file not found, network service (temporarily) overloaded, malformed email

When there is no bug or error, the program will produce a **result**, which is an indication of the outcome/status of an operation.

Handling Techniques: Overview

- **roll your own**: The programmer must "roll their own" handling, like defining enumerated types (success,error) to communicate results. <!-- - **error objects**: used to return an explicit error result from a function to its caller, independent of any valid return value.
- **optional objects**: used by a function to return a single result that can represent either a valid value or a generic failure condition.
- **result objects**: used by a function to return a single result that can represent either a valid value or a specific Error Object.
- **assertions/conditions**: checks whether a required condition is true, and immediately exits the program if it is not.
- **exceptions/panics**: f() may "throw an exception" which exits f() and all calling functions until the exception is explicitly "caught" and handled by a calling function or the program terminates. -->

Error Objects

- language-specific objects used to return an explicit error result from a function to its caller, independent of any valid return value.
- error objects are returned along with a function's result as a *separate return value*.
- Languages with error objects provide a *built-in error class* to handle common errors
- you can also define custom error classes with fields that are specific to your error condition, and even wrap (nested) errors to provide more context.

Optionals

- "Optional" object can be used by a function to return a single result that can represent either a valid value or a *generic failure*.
- like ADT that contains either "nothing", or "something" (with a value)

- only use an Optional if there's an obvious, single failure mode.

In some languages, optionals are a built-in part of the language with dedicated syntax! Swift:

```
func divide(a: Float, b: Float) -> Float? {
    if b == 0 {
      return nil;
    }
    return a/b;
}

var opt: Float?;
opt = divide(a:10, b:20);

if opt != nil {
  let a_div_b: Float = opt!;
  print("Result was: ", a_div_b);
} else {
  print ("Error result!")
}
```

- the `?` as in `Float?` is used for optional return value.
  - `nil` is used for error result
  - any other value directly creates the optional that represents a valid value.
- `!` is used to extract the value from the optional if it is valid.

Result Object

- "Result" object returns single result that can represent either a valid value or a *distinct error*.
- think of a Result as a struct that holds two items: a value and an Error object with details about the error
- can use it when there are multiple distinct failure modes that need to be distinguished and handled differently.

```
enum ArithmeticError: Error {
  case divisionByZero
  // add other error types here as necessary
}

func my_div(x: Double, y: Double) -> Result<Double, ArithmeticError> {
  if y == 0  { return .failure(.divisionByZero) }
  else       { return .success(x / y) }
}

let result = my_div(x:10, y: 0)
switch result {
  case .success(let number):
    print("Successful division: ", number)
  case .failure(let error):
```

```
    dealWithTheError(error)
}
```

Assertions

- **assertion** - tests a particular condition and terminates the program if it's not met
  - states what you expect to be true for correct execution
- *program aborts if it's not true* -- this is because it can't correctly continue if assertion isn't true

Typically used to verify:

- **preconditions**: something that must be true at the start of a function for it to work correctly.
- **postconditions**: something that the function guarantees is true once it finishes.
- **invariants**: a condition that is expected to be true across a function or class's execution.

Consider the states to verify in selection sort: `void selection_sort(int *arr, int n);` .

- preconditions: `arr` must not be `nullptr`, `n` must be `>= 0`
- postconditions: For all `i`, `1 <= i < n`, `arr[i] > arr[i-1]`
- invariants: At the end of iteration `j`, the first `j` items of `arr` are in ascending order

```
// C++ assertions
void selection_sort(int *arr, int n) {
  assert(arr != nullptr);
  assert(n >= 0);
  ...
}
```

When to use assertions instead of error object, optional, result?

- When there is a logic error that must be true for the program to work

Exception Handling

- With other approaches, error checking is woven directly into the code--harder to understand the core business logic
- With exception handling, separate the handling of exceptional situations/unexpected errors from the core logic
- errors are communicated and handled independently of the mainline logic
- more readable code that focuses on logic, isn't littered with extraneous error checks

Participants of Exception Handling

catcher:

- A block of code that "tries" to complete one or more operations that might result in an unexpected error.
- An "exception handler" that runs if (and only if) an error occurs during the tried operations, and deals with the error.

Thrower

- A function that performs an operation that might result in an error
- If an error occurs, the thrower creates an exception object containing details about the error, and "throws" it to the exception handler to deal with.

Execution Flow

- If any operation in the try block throws an exception, the exception is immediately passed to the exception handler (in catch block) for processing.
- remaining statements in the try block are skipped
- when the exception handler completes, execution continues starting from after the catch block
- if there is no exception, the `catch` is skipped

```
void f() {
  do_thing0();
  try {
    do_thing1();
    do_thing2();  // possible exception
    do_thing3();
  }
  catch (Exception &e) {
    deal_with_issue(e);
  }
  do_post_thing();
}
```

Exception generated in `do_thing2()`:

- `do_thing0`
- `do_thing1`
- `do_thing2`
- `deal_with_issue`
- `do_post_thing`

No exception:

- `do_thing0`
- `do_thing1`
- `do_thing2`
- `do_thing3`
- `do_post_thing`

What is An Exception?

- **exception** - an object with one or more fields which describe an exceptional situation.
- At a minimum, every exception has a way to get a description of the problem that occurred, e.g., "division by zero."
- programmers can also use subclassing to create their own exception classes and encode other relevant info
- Note: exceptions very slow when thrown

Nested Exceptions

- an exception may be thrown an arbitrary number of levels down from the catcher.

- dangerous! functions in the middle that don't catch the exception are ended and things could be left in bad state (eg: memory leaks, leftover temp files)

The Exception Hierarchy

- exception handler can specify exactly what *type* of exception(s) it handles
- A thrown exception will be directed to the *closest* handler (in the call stack) that *covers its exception type*

```
void h() {
  throw out_of_range("negative index");
}

void g() {
  try {
    h();
  }
  catch (invalid_argument &e) {
    deal_with_invalid_argument_err(e);
  }
}

void f() {
  try {
    g();
  }
  catch (overflow_error &e) {
    deal_with_arithmetic_overflow(e);
  }
  catch (out_of_range &e) {
    deal_with_out_of_range_error(e);
  }
}
```

- if an exception is thrown for which there is no compatible handler, the program will just *terminate*
  - equivalent to a "panic" which basically terminates the program when an unhandle-able error occurs.

Derived Exceptions

- different types of exceptions are derived from more basic types of exceptions
  - C++ has an exception hierarchy, where `std::exception` is the base class of all exceptions
- If you want to create a catch-all handler, you can have it specify a (more) basic exception class. The handler will deal with the base exception type and all of its subclasses

Finally

- `finally` block guaranteed to run whether the try block succeeds or throws
  - Java
- enables you to put cleanup code in a single place, guaranteed to run in either case

```java
// Java "finally" block
public class FileSaver {
  public void saveDataToFile(String filename, String data) {
    FileWriter writer = null;
    try {
      // Next 2 lines might throw IOExceptions
      writer = new FileWriter(filename);
      writer.write(data);
    }
    catch (IOException e) {
      e.printStackTrace();  // debug info
    }
    finally {
      if (writer != null)
        writer.close();
    }
  }
}
```

Exceptions and memory safety

- Exceptions can create all sorts of bugs if not used appropriately

```
void h() {
  if (some_op() == failure)
    throw runtime_error("deets");

  do_other_stuff();
}
void g() {
  int* arr = new int[100];
  h();
  // code uses array here...
  delete[] arr;
}
void f() {
  try {
    g();
  }
  catch (Exception &e) {
    deal_with_issue(e);
  }
}
```

- if `h` throws a `runtime_error`, the memory allocated in `g` is never freed, thus causing a memory leak!
- Should be wary of using exceptions in languages *without* automatic memory management

Exception Annotations

- Some languages force you to annotate functions which throw exceptions with a special token

    - Java: must annotate functions that throw with the `throws` keyword
- explicitly tells if you need to catch an exception when you call a function.

Exception handling guarantees

For safety, when writing a function, always make sure you meet one of the following guarantees:

- *No throw guarantee*: A function guarantees it won't throw an exception
    - If an exception occurs in/below the function, it will handle it internally, and not throw. For example, this is required of destructors, functions that deallocate memory, swapping functions, etc.
- *Strong exception guarantee*: If a function throws an exception, it guarantees that the program's state will be "rolled-back" to the state just before the function call
    - eg: `vec.push_back()` ensures the vector doesn't change if `push_back()` fails.
- *Basic exception guarantee*: If a function throws an exception, it leaves the program in a valid state (no resources are leaked, and all invariants are intact)
    - state may not be rolled back, but at least the program can continue running

Panics

- Like an exception, a panic is used to abort execution due to an exceptional situation which cannot be recovered from (e.g., an unrecoverable error).
- Essentially, a panic is an exception which is never caught
- Panics contain both an error message and a stack trace to provide the programmer with context as to why the software failed.

```
// C# FailFast panic
class WorstCaseScenario
{
  public void someFunc()
  {
    if (somethingUnrecoverableHappens())
      Environment.FailFast("A catastrophic failure has occurred.");
  }
}
```

Why would you use exception handling over other approaches

- when you want to make sure the exception is dealt with
- use other approaches for expected errors
- exception should be used for rare errors or bugs

Best Practices

Assertions

- to check for errors that should never occur if code is correct (bugs, unmet preconditions, violated invariantes)
- to build unit tests

Optional, Error, Result Object

- function needs to return either a value OR an error for common, recoverable failure case
  - optional if only one failure mode
  - error/result object when multiple failure modes possible and you need to pass details to caller

Exceptions

- when a funciton is unable to fulfull its "contract" AND it's an error that occurs rarely (<1%) AND failure can be recovered from

Panics

- when there's no reasonable way to recover from an error

First Class Functions

- Functions can be passed/returned to/from other functions
- Variables can be assigned to functions
- Functions can be stored in data structures
- Functions can be compared for equality
- Functions can be expressed as anonymous, literal values

Functions here are called "first-class citizens" because they are data objects and can be maniupulated like any other data in a program.

Here are some examples of first class functions in some languages:

- In C++, first-class functions are implemented with function pointers.

```
rtype (*) (type1 param1, type2 param2, ...)
```

```
int square(int val) { return val * val; }
int fivex(int val) { return val * 5; }

using IntToIntFuncPtr = int (*)(int val);

void apply(IntToIntFuncPtr f, int val) {
  cout << "f(" << val << ") is " << f(val) << endl;
}

IntToIntFuncPtr pickAFunc(int r) {
  if (r == 0) return square;
         else return fivex;
}

int main() {
 IntToIntFuncPtr f = pickAFunc(rand() % 2);
  if (f == square) cout << "Picked square\n";
               else cout << "Picked fivex\n";
  apply(f, 10);
}
```

- Go

```
func square(val int) int { return val * val }
func fivex(val int) int  { return val * 5 }

// type alias
type IntToIntFuncPtr func(int) int

func apply(f IntToIntFuncPtr, val int) {
  fmt.Println("f(", val, ") is ", f(val))
}

func pickAFunc(r int) IntToIntFuncPtr {
  if r == 0 {
    return square
  } else {
    return fivex
  }
}

func main() {
  var f = pickAFunc(rand.Intn(2))
  if f != nil {
    apply(f, 10)
  }
}
```

Anonymous (Lambda) Functions

- a function with no name, anonymous
- typically passed as an argument to another function or stored in a variable
- used when a short, temporary function is needed just once in a program, and it doesn't make sense to define a general-purpose named function.

Every lambda has three different parts:

- Free Variables to Capture - What variables defined outside the lambda function should be available for use in the lambda function when it is later called.
- Parameters & Return Type - What parameters does the lambda function take and what type of data does it return.
- Function Body - The body of the lambda function that performs its operations.

Lambdas in other languages (not Haskell) don't need to be pure, can modify oustide state

Capture by Value

- only the values of the free variables are captured by the lambda
- any changes made to the captured variable inside the lambda won't be reflected outside

```
// C++
[captured_var1, ...](type1 param1, ...) -> rtype {/* body */}
```

```
// C++ lambda function
auto create_lambda_func() {
```

```
  int m = 5;
  int b = 3;
  return [m,b](int x) -> int { return m*x + b; };
}

int main() {
  auto slope_intercept = create_lambda_func();
  cout << "5*100 + 3 is: " << slope_intercept(100);
}
```

Capture by Reference

```
// C++ lambda function – capture by reference
auto fun_with_lambdas() {
  int q = 0;
  auto changer = [&q](int x) { q = x; };
  changer(5);
  cout << "q is now: " << q; // Outputs "q is now: 5"
  return changer;
}

int main() {
  auto f = fun_with_lambdas();
}
```

Capture by Environment

In capture by environment, an object reference to the *lexical environment* where the lambda was created is added to the closure. The *lexical environment* is a data structure that holds a mapping between every in-scope variable and its value. That includes all variables in the current activation record (locals, statics), and all global variables. Python uses capture by environment semantics. Here is an example.

```
def foo():
  q = 5
  f = lambda x: print("q*x is: ", q*x)
  f(10)
```

When you define the lambda, it creates a closure containing:

- the lambda function itself
- an object reference to the current lexical environment

When running the lambda, it looks up each free variable in the lexical environments to obtain its value. Now consider this slightly modified code.

```
def foo():
  q = 5
  f = lambda x: print("q*x is: ", q*x)
  f(10) # outputs "q*x is: 50"
  q = q + 1
  f(10) # outputs "q*x is: 60"
```

This outputs both `50` and `60`. This is because even though the lambda was defined when `q = 5`, when the lambda runs, it looks up the latest value of the variable in the lexical environment, and therefore outputs `60`.

Polymorphism

- **Polymorphism** a technique where we define a function or class that is able to operate on multiple different types of values
- goal is to express algorithms with minimal assumptions about the types of data they operate on, making them as interoperable as possible - ideally without losing performance
  - Ex: rather than creating a class for a linked list of strings, and a separate class for a linked list of ints, create a single polymorphic linked list class for any value
- generally implemented differently in statically and dynamically-typed languages

Subtype Polymorphism

- a function is designed to operate on objects of a base class B (e.g., Shape) and on objects of all subclasses derived from B (e.g. Circles)
- covered in depth in OOP

```
class Shape {...}
class Square: public Shape {...}
class Circle: public Shape {...}

void processShape(Shape& s)
 { ... }

Circle c(10);
processShape(c);
Square s(5);
processShape(s);
```

Ad-hoc Polymorphism

- define specialized versions of a function for each type of object we wish it to support (overloading)
- language decides which version of the function to call based on the types of the arguments
- note: not possible in dynamically typed languages as we don't specify types for formal parameters, so can't define multiple versions of a function with different parameter types
  - specialise behaviour using type reflection

```
bool greater(Dog a, Dog b) {
  return a.bark() > b.bark();
}
bool greater(Student a, Student b) {
  return a.gpa() > b.gpa();
}
```

Parametric Polymorphism

- define a single, parameterized version of a class or function that can operate on many, potentially unrelated types

Templates

- do almost all of the work at compile-time
- each time you use the template with a different type the *compiler* generates a concrete version of the template function/class by substituting in the type parameter
- compiles the newly-generated functions/classes as if they were never templated in the first place
    - Ad-hoc polymorphism under the hood
- Any operations you use in your templated code must be supported by the types being templated
- templates are *type-safe* because a concrete version is generated and compiled
- also means templated code is just as efficient
- note: C-style macros are sort of the OG templates. The pre-processor would basically do a textual search-and-replace of the arguments, the compiler doesn't generate a new function for each parameterized type.

```
template <typename T>
void takeANap(T &x) {
  x.sleep();
}
class Dog {
public:
  void sleep() { ... }
};

class Person {
public:
  void sleep() { ... }
};
...
int main() {
  Dog puppers;
  takeANap(puppers); // OK!

  Person carey;
  takeANap(carey); // OK!

  string val;
  takeANap(val); // error: no member named 'sleep' in 'string'
}
```

```
// C macros used to implement template-like functionality

#define swap(T  ,a,b) { T   temp; temp = a; a = b; b = temp; }

int main() {
 int p = 5, q = 6;
```

```
 swap(int,p,q);

 std::string s = "foo", t = "bar";
 swap(std::string, s, t);
}
```

Generics

- compile just one "version" of the generic function or class - independent of the types that actually use our generics.
- Because of this, the generic can't make assumptions about what types it might be used it (**type-agnostic**), only do "generic operations"
    - comparing object references
    - assignment
    - instantiation
- type checking performed once generic is parametrized
    - if you parametrize a generic with type A, you can't use with type B

```
// C# generic container
class HoldItems<T> {
  public void add(T val)
    { arr_[size_++] = val; }
  public T get(int j)
    { return arr_[j]; }
 public void beADuck(int j)
    { arr_[j].quack(); } // ILLEGAL!!

  T[] arr_ = new T[10];
  int size_ = 0;
}

HoldItems<Duck> ducky = new HoldItems<Duck>();
ducky.add(new Duck("Daffy"));
Duck d = ducky.get(0);
Robot r = ducky.get(0);   // ERROR
```

- **unbound generics** - no bounding on operations, can only use most general
- **bound types** - add restrictions on what types are allowed
    - can use any operation the bound type can perform

```
// C#
interface DuckLike {
  public void quack();
  public void swim();
}

class HoldItems<T> where T: DuckLike {
  public void add(T val)
    { arr_[size_++] = val; }
  public T get(int j)
    { return arr_[j]; }
 public void beADuck(int j)
```

```
    { arr_[j].quack(); } // LEGAL!!

  T[] arr_ = new T[10];
  int size_ = 0;
}
```

```haskell
-- haskell
qsort :: (Ord t) => [t] -> [t]
qsort [] = []
qsort (x:xs) =
  let leq = qsort [j | j <- xs, j <= x]
      geq = qsort [k | K <- xs, k > x]
  in leq ++ [x] ++ geq
```

Now we finally know what the `Eq`, `Ord`, etc. type classes mean - they place *bounds* on what types the `t` type variable can take on.

Why Parametric Polymorphism?

Couldn't we just use inheritance?

Here, *just* inheritance fails. For context, languages like C# and Java have an `Object` class that superclasses *all other* classes. Let's assume we can do polymorphism just with subclassing:

```
class Duck : Object {
  public void quack() {
    Console.WriteLine("Quack");
  }
  public void swim() {
    Console.WriteLine("Paddle");
  }
}

class HoldItems {
  public void add(Object val)
    { arr_[size_++] = val; }
  public Object get(int j)
    { return arr_[j]; }

  Object[] arr_ = new Object[10];
  int size_ = 0;
}
```

Any object can be stored in `HoldItems`! So, we could add both a `string` and a `Duck` to a container, since both subclass `Object`.

- no way for us (or C#, at compile-time) to detect a bug!

With generics, we tell the compiler which types our generics will use:

```
HoldItems<Dog> items = new HoldItems();
items.add(new Duck("Daffy")); // type error!!
```

Templates vs Generics

- Generics are more conservative, since they can only use type-agnostic functionality
- templates can use any functionality, so long as the templated type is compatible
- bounded generics let you use more specific functionality but can only parameterize your with types that are subtypes (or the same type) as the bounded type

Specialization

- defining a dedicated version of a function/class for a specific type, used in place of generic version of that function/class
- enables custom versions to handle special cases or improve efficiency

```
// optimized boolean arrays
template<>
void sort(bool items[], int len)
{
  int n_false = 0;
  for (int i=0;i<len;++i) { if (!items[i]) ++n_false; }
  for (int i=0;i<n_false;++i) items[i] = false;
  for (int i=n_false;i<len;++i) items[i] = true;
}

template <typename T>
void sort(T items[], int len)
{
  // General version of the sort
}
```

Parametric Polymorphism in Dynamically Typed Languages

- can't *really* have "type-parameterized" classes or functions, since variables don't specify types!
- Note: duck typing occupies the same feature space
    - but isn't as safe - we have no idea at compile-time if it'll work!